Обзор масштабируемых систем межмодульных оптимизаций

Долгорукова К.Ю.
<unerkannt@ispras.ru>
ИСП РАН, 109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. Большинство приложений имеют модульную структуру, но оптимизация таких программ при сборке по раздельной схеме “компиляция-связывание” ограничивается отдельным модулем. И, хотя многие компиляторы поддерживает межмодульные оптимизации, в случае больших приложений их полноценное проведение зачастую неприемлемо ввиду значительных затрат времени и памяти. Решающие эту проблему компиляторные системы, способные производить межмодульные оптимизации с учетом возможностей аппаратуры и требований пользователя по затратам ресурсов, называются масштабируемыми системами межмодульных оптимизаций, или оптимизаций времени связывания.

В данной статье мы ставим задачу рассмотреть прежде всего различные подходы к проблеме масштабируемости инфраструктуры относительно потребляемых ресурсов, а не сами межпроцедурные и межмодульные оптимизации или их эффективность в той или иной реализации, так как они по большому счёту не зависят от используемой инфраструктуры. Интерес представляют возможности распараллеливания тех или иных стадий компиляции целой многомодульной программы, а также способы экономии памяти при межмодульном анализе.

Данный обзор включает в себя несколько компиляторных систем для языков общего назначения: C, C++, Fortran, – но нередко системы способны оптимизировать и другие языки, если для них реализован соответствующий генератор промежуточного представления.

В заключении статьи будут представлены предварительные результаты по масштабированию компонентов связывания инфраструктуры LLVM.

Ключевые слова: компиляторы; межпроцедурные оптимизации; межмодульные анализ и оптимизации; системы межмодульных оптимизаций.

1. Введение

Б*о*льшая часть инфраструктур межпроцедурных оптимизаций имеет в своей основе общую модель компиляции, в которой представлены 3 основные фазы: генерация промежуточного кода (ГПК), фаза межпроцедурных оптимизаций и генерация машинного кода (ГМК).

Во время ГПК компилятор получает файлы с кодом на исходном языке программирования, проводит небольшое количество локальных оптимизаций в каждом файле отдельно, а также генерирует дополнительную информацию о зависимостях между функциями в так называемые summary, или аннотации, представляющие собой некую информацию о программе, необходимую для проведения межпроцедурного анализа. В результате работы первого этапа генерируется расширенный файл в промежуточном представлении (либо в двоичном формате), содержащий также дополнительную информацию. Обычно этап ГПК легко параллелизуем.

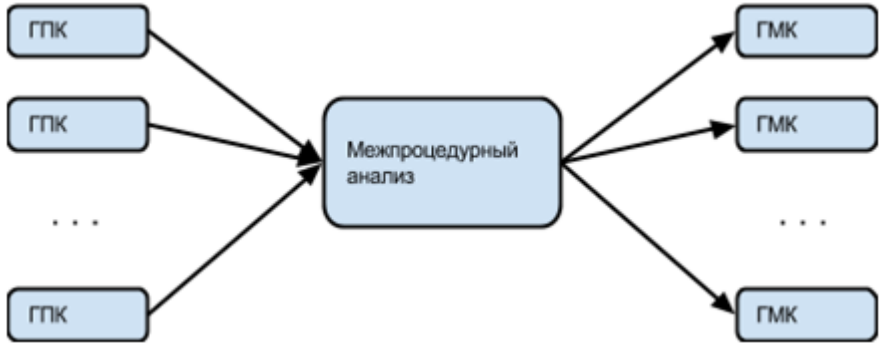

Рис. 1. Схема работы систем оптимизаций времени связывания

Второй этап как правило проходит во время связывания и читает файлы, полученные на первом этапе, анализирует межпроцедурные зависимости, проводит некоторые оптимизации, и снова генерирует файлы в промежуточном представлении и некоторое количество дополнительной информации. Этот процесс трудно параллелизуем, и все компиляторные системы с открытым исходным кодом выполняют межмодульный анализ в один поток, поэтому именно анализ – узкое место этих систем.

Последняя фаза принимает файлы, сгенерированные на втором этапе и генерирует из них объектный код, который затем подается ассемблеру. Иногда на этой стадии проводятся дополнительные оптимизации. Эта фаза у разных компиляторов может быть параллелизуемой или непараллелизуемой, и в результате может получаться либо один объектный файл, либо несколько, которые впоследствии передаются стандартному системному компоновщику.

Несмотря на то, что развитие многих представленных в статье систем проходило примерно в одно время – и разработка большинства из них ведется до сих пор, – мы постараемся рассмотреть эти системы в хронологической последовательности относительно их появления, делая исключения лишь для наследуемых один от другого систем для простоты сравнения. Так, несмотря на относительную “свежесть” SYZYGY, мы рассмотрим его вторым из-за сильной родственной связи с HLO. Затем мы рассмотрим Open64 и

межпроцедурный компилятор из GCC. В следующей главе речь пойдет о LLVM, как об альтернативном взгляде на межмодульные оптимизации. Последним рассматривается легковесный компилятор от Google, который учёл все успехи и ошибки предшественников и стоит особняком от остальных представителей систем межпроцедурных оптимизаций из-за радикальности подхода.

Для того, чтобы не повторяться, говоря о похожих сущностях, в каждой посвященной какому-то определенному фреймворку главе мы будем говорить только о нововведениях, а при необходимости будем ссылаться на реализованные ранее подходы.

2. HLO

High-Level Optimizer (оптимизатор высокого уровня) [1] – одна из первых промышленных платформ с возможностью масштабирования по памяти. Ограничение памяти достигается, прежде всего, сохранением неиспользуемых в текущий момент данных оптимизатора на диск.

Авторы системы протестировали её на скомпонованных без оптимизаций, с межпроцедурными оптимизациями, оптимизациями с профилем и межпроцедурными оптимизациями с профилем крупных тестах SPECint95 и ISV Apps, и пришли к выводу, что самый большой положительный скачок (до 71%) производительности программам даёт именно совместное использование межпроцедурных оптимизаций и профиля. Поэтому авторы системы проектировали HLO с ориентацией именно на такие оптимизации[2].

HLO проектировался как компонент компилятора HP-UX, схема работы которого представляет собой конвейер и показана на рисунке ниже.

Внутри HLO используется система проходов, в которых есть проходы как анализа, так и преобразований, при этом есть и внутрипроцедурные, и межпроцедурные проходы. Но все они, как легко понять из названия оптимизатора, машинно-независимые. Внутри LLO (Low-Level Optimizer – низкоуровневый оптимизатор) также по принципу проходов производятся машинно-зависимые оптимизации.

Инструментирование производится на межпроцедупные ветви и вызовы функций, при этом в компиляторе есть возможность выбрать, на какой стадии его внедрять. После запуска инструментированной программы генерируется база данных, при этом в задачу компилятора входит поддержка соответствия собранного профиля коду исполняемой программы.

Для возможности межмодульной оптимизации ГПК сохраняет промежуточное представление в объектные файлы, соответствующие компилируемым модулям. Когда компоновщик считывает все объекты промежуточного представления, он передает их оптимизатору и генератору кода.

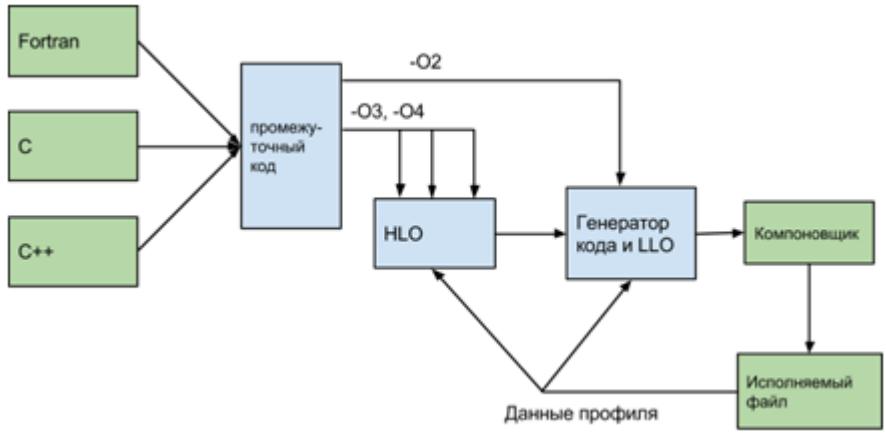

Рис. 2. Схема работы компилятора HP-UX

2.1 Модель “не всё в памяти”

Так как потребление памяти при межмодульных оптимизациях больших программ может быть невероятно огромным (до 1.7 кб на каждую строку исходного кода!), разработчики из HP предложили модель NAIM (Not-All-In-Memory – “не всё в памяти”), позволяющую частично сбрасывать данные на диск для ресурсоёмких оптимизаций.

Снижение потребления достигается несколькими методиками: сжатием структур, классификацией данных на используемые и неиспользуемые в данный момент, и выгрузкой объектов из памяти вместе со сбросом на диск.

Для сжатия объекты классифицируются как *глобальные, временные* и *производные*. *Глобальные* объекты всегда находятся в памяти и представляют собой такие структуры, как граф вызовов, таблица символов всей программы, и таблица модулей компиляции, в которой хранятся ссылки на объекты временных структур. Временные структуры – это таблицы символов для каждого модуля и процедуры в промежуточном представлении.

Все *временные* объекты могут находиться в трёх состояниях: в развернутом виде в памяти, в сжатом виде в памяти, либо в сжатом виде сброшенные на диск.

Третья категория содержит объекты, которые всегда можно сгенерировать на основе временных и глобальных объектов. К этим объектам относятся рёбра потоков данных, деревья интервалов, аннотации переменных индукции. *Производные* объекты не нужно сбрасывать на диск, достаточно их удалить, а при необходимости сгенерировать заново. Организация объектов показана на рис.2.

Сам механизм выгрузки объектов работает так: данные, необходимые для выполнения текущего прохода, загружаются в память, а после его выполнения ненужные уже структуры упаковываются и сбрасываются на диск, либо помечаются для удаления. Во время упаковки объекты приводятся к независящей от указателей форме, для этого каждому объекту назначается идентификатор (PID – persistent identifier). Во время распаковки каждый PID преобразовывается в адрес.

Для более эффективного управления памятью в HLO используется динамическое размещение в памяти и сборщик мусора. При этом, менеджер памяти старается учитывать локальность кода и складывать зависимые объекты близко друг к другу.

Сама же модель “не всё в памяти” используется не всегда. В частности, если имеет место оптимизация маленьких приложений, то программа загрузится в память целиком. Для включения разных механизмов NAIM, потребление памяти должно перейти несколько границ, зависящих от свойств аппаратного обеспечения и параметров пользователя. По достижении первой границы, система сначала включает механизм упаковки тел функций, затем, при переходе через вторую отметку, начинают упаковываться таблицы символов. Если же количество потребляемой памяти перешагнет и через третий порог, включится механизм выгрузки функций. Выгрузка работает “ленивым” образом, то есть объекты начинают выгружаться, когда система займёт всю отведённую ей память.

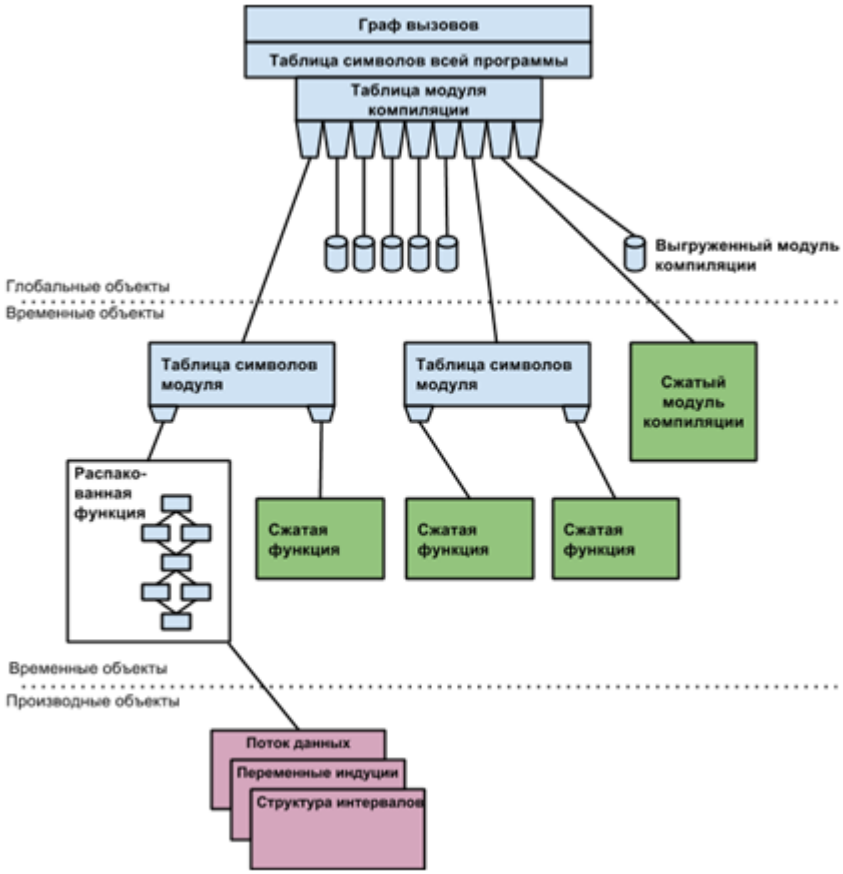

Рис. 3. Схема "не всё в памяти" компилятора HLO

2.2 Выбор участков кода для оптимизации и профилирование

Было замечено, что для получения максимальной производительности на деле достаточно оптимизировать не более 20% кода[12]. Чтобы не оптимизировать участки кода, которые заведомо в этом не нуждаются, в системе реализованы механизмы крупнозернистой и мелкозернистой выборки. Крупнозернистая выборка просматривает данные профиля и выбирает наиболее горячие модули для оптимизации. Для этого компилятор сортирует вызовы по частоте согласно профилю в убывающем порядке. Далее отбираются вызовы, набравшие заданный пользователем процент, производится поиск модулей, содержащих вызывающие и вызываемые функции – и к ним применяются межмодульные оптимизации с

профилем. К остальным применяются обычные внутримодульные оптимизации с профилем.

При мелкозернистой выборке межмодульные оптимизации производятся над горячими функциями, независимо от содержащих их модулей.

Сбор профиля производится посредством встраивания, при этом даже при изменении исходного кода есть возможность использовать старый профиль повторно.

Система межмодульной оптимизации HLO была внедрена в компиляторную систему HP-UX начиная с версии 9.0, модель “не всё в памяти” была добавлена в неё начиная с версии 10.20. Прирост производительности сгенерированного системой кода составлял от 20 до 70% относительно обычной компиляции без межмодульных оптимизаций.

3. SYZYGY

Компилятор SYZYGY[3] – прямой последователь HLO, он также был разработан как компонента для системы HP-UX. Но в данном компиляторе уже используется модель оптимизации из двух стадий: решение об оптимизациях и сами оптимизации. В целях ограничения потребляемой памяти фреймворк использует алгоритмы, которые поддерживают открытыми не более некоторого числа файлов с промежуточным представлением одновременно, и минимизируют количество открытий и закрытий файлов. После оптимизаций промежуточное представление записывается в файлы обратно, и на них прогоняется параллелизуемый ГМК.

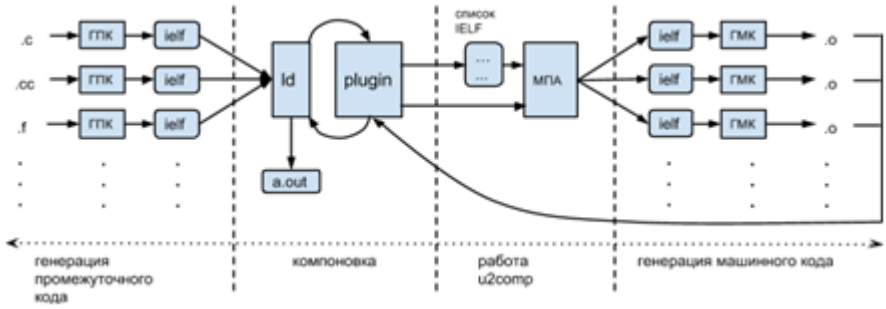

Рис. 4. Схема работы SYZYGY

Сбор данных о процедурах и модулях для аннотаций происходит во время работы генерации промежуточного кода. Перед сбором данных ГПК производит над кодом такие оптимизирующие преобразования, как свёртка констант, канониканизацию промежуточного представления, упрощение алгебраических выражений. Это необходимо для уменьшения размеров и увеличения точности аннотаций.

Стадия межпроцедурного анализа (МПА) строит таблицу глобальных символов, производит разрешение имён и унификацию типов, а также строит граф вызовов.

Оптимизации, которые требуют контекст из других модулей, – например, встраивание функций, клонирование процедур и распространение неявных вызовов, – разделены в SYZYGY на две стадии. Первая стадия производится во время анализа, вторая – во время генерации машинного кода (ГМК). Так как ГМК работает только с одним файлом за раз, необходимая информация о функциях в других файлах выносится в промежуточное представление во время стадии анализа. Во время стадии ГМК производятся только внутримодульные оптимизации и генерация кода.

3.1 IELF

Промежуточное представление хранится в формате, представляющем собой обычный 32 или 64-битный файл формата ELF[4], включающий заголовок, таблицу символов, таблицу строк, а также несколько секций внутреннего представления скомпилированной программы в двоичном виде.

В памяти промежуточное представление сохраняется в областях – блоках памяти, управляемых компилятором. Ссылки в областях представляются идентификаторами, а не указателями, для этого у каждой области есть массив идентификаторов. В файле IELF этот массив хранится в отдельной секции, также есть секция отображений для быстрого поиска этого массива. Считывание массива идентификатора происходит во время чтения или записи файлов, когда это необходимо. Во время работы анализатора в памяти поддерживается всего несколько файлов, остальные сбрасываются на диск. Обычно размер файла в формате IELF в среднем в пять раз больше, чем обычный объектный файл, но в худшем случае этот показатель может достигать 700. Это связано с тем, что в файле должна содержаться информация о всех используемых типах. Если, например, объектный файл состоит из нескольких строк, в одной из которых используется ссылка на поле объекта очень большого класса из другого файла, то вся необходимая информация о типах будет указана и в файле, ссылающемся на объект.

3.2 Схема работы

Модель компиляции SYZYGY тоже состоит из стадий: генерация промежуточного кода, анализ и генерация машинного кода, – и содержит соответствующие компиляторные компоненты. ГПК и ГМК выполняют обработку одного файла за раз и могут быть запущены в параллельных потоках. Анализатор работает со всеми файлами в один поток.

ГПК получает на вход файлы исходного кода и генерирует из них IELF файлы после применения немногочисленных простых оптимизаций. Во время связывания компоновщик, представленный утилитой ld, разрешает

зависимости, после чего передает эстафету плагину, который собирает список файлов формата IELF, распаковывает некоторые из этих файлов и объединяет с информацией, полученной от компоновщика. Затем плагин вызывает программу u2comp, передавая ей список IELF-файлов и собранную компоновщиком и плагином информацию. u2comp – это утилита, производящая межпроцедурный анализ и некоторые преобразования в промежуточном представлении. После окончания всех вышеперечисленных действий она генерирует новый набор IELF-файлов, а затем файл инструкций сборки “makefile”, в которых в многопоточном режиме запускается ГМК.

3.3 Межпроцедурные оптимизации

Как было отмечено ранее, такие оптимизации, как встраивание, разделены, так как требуют дополнительных подготовительных действий. Встраивание функций может сильно изменить структуру программы, поэтому, чтобы избежать изменений в аннотациях, оно проводится в самом конце фазы анализа.

Программа u2comp начинает работу с построения глобальных структур данных, таких как таблица глобальных символов. Затем происходит удаление лишних секций COMDAT, содержащих мёртвый код и дублирующие участки кода – и изменения фиксируются в файлах. Удаления COMDAT-секций необходимо для уменьшения размера IELF-файлов, а значит, и для облегчения последующего чтения файлов.

Затем происходит несколько межпроцедурных оптимизаций. Необходимые для них результаты анализа либо содержатся в памяти, либо сбрасываются в файлы, в зависимости от размера. Чтобы не переписывать после каждой оптимизации анализ, изменения, произведенные оптимизирующими преобразованиями, записываются в список изменений, а перед самой процедурой встраивания они применяются одна за другой к промежуточному представлению программы. Для облегчения обновления аннотаций также поддерживается список “заплаток”, который потом разом применяется к промежуточному представлению во время очередного чтения соответствующего участка кода.

Во время самого ресурсоёмкого, но весьма эффективного преобразования – встраивания, – программа не держит всё промежуточное представление в памяти. Также в постоянной памяти находится лишь незначительный набор аннотаций, или ядро аннотаций. Количество одновременно открытых файлов определяется потребляемой ими памятью. Если программа достигла лимита предоставленной ей памяти, она начинает закрывать неиспользуемые файлы перед открытием новых. В SYZYGY всего одна фаза анализа и одна – преобразований, а значит, встраивание проводится всего один раз. Поэтому аннотации обновляются после каждого решения о встраивании.

В сравнении со своим предшественником, HLO, SYZYGY работает быстрее в среднем в 2 раза, показывая похожие результаты производительности. Для однопоточного запуска издержки составляют 2.3x, а для запуска в 4 потока – 1.2x на SPEC2000.

4. Open64

Open64 – небольшая исследовательская компиляторная система, изначально созданная для работы на процессорах семейства Intel, что, впрочем, не мешает ей работать и на многих современных архитектурах[5]. Мы поверхностно упомянем применяемые в нем методы, а затем глубже разберем его прямого и более развитого и документированного потомка – WHOPR из GCC.

Open64 имеет стандартную структуру, то есть содержит параллелизуемые ГПК, ГМК и непараллелизуемый межпроцедурный этап. Во время работы ГПК создаются аннотации, и записывается вместе с промежуточным представлением WHIRL в файл. Межпроцедурный этап содержит фазы анализа и оптимизирующих преобразований. На фазе анализа компилятор работает только с аннотациями и не открывает промежуточное представление, поэтому она проходит быстро относительно предшественников. На фазе оптимизации компилятор уже работает с промежуточным представлением файлов, загружая их все в память сразу. Затем он записывает оптимизированную программу в объектный файл формата ELF, причём, количество выходных файлов может не соответствовать количеству входных: компилятор предварительно разбивает программу в зависимости от размера и внутренних зависимостей процедур. Open64 производит также один промежуточный объектный файл, содержащий все статически “продвинутые” и глобальные переменные, а также межпроцедурную таблицу символов и типов данных. Во время ГМК этот файл обрабатывается первым, перед параллельной компиляцией остальных промежуточных файлов. После этого на полученных объектные файлах запускается ГМК[6].

5. GCC

В GCC режим масштабируемых межмодульноых оптимизаций называется WHOPR – WHole Program Optimization[7]. WHOPR отчасти похож на Open64, но в нём также реализованы такие новые особенности, как виртуальные клоны и функции перехода, а также компилятор способен классифицировать функции по частоте использования согласно профилю.

Дополнительные секции, добавляемые в ELF во время ГПК (фаза *LGEN* – Local Generation, локальная генерация) – это опции командной строки (.gnu.lto_.opts), таблица символов (.gnu.lto_.symtab), глобальные типы и объявления (gnu.lto_.decls), граф вызовов (.gnu.lto_.refs), тела функций в промежуточном представлении, инициализаторы статических переменных (.gnu.lto_.vars) и, собственно, аннотации.

Во время второй фазы (*WPA* – Whole Program Analysis) работа ведется без доступа к телам функций и инициализаторам переменных, то есть на основе аннотаций. И обновляется в файлах только участки с аннотациями, куда записываются решения об изменении функций.

ГМК (в GCC он называется *LTRANS* – Local Transformation, локальное преобразование), – в отличие от Open64, стремящемся целиком записать граф вызовов в один файл, когда это позволяет память, – работает над файлами, тела которых остаются в там же, за исключением, быть может, мёртвых и полностью встроенных функций в промежуточном представлении. Фаза ГМК производит локальные для файлов оптимизации на основе информации, находящейся в секциях с аннотациями, в параллелизуемом режиме.

5.1 Виртуальные клоны и функции перехода

Отличительной особенностью компилятора GCC является наличие инфраструктуры так называемых виртуальных клонов, необходимой для поддержания целостности функций во время преобразований. Виртуальные клоны в графе вызовов – это функции без тел, имеющие только описание, как создать эту функцию на основе другой функции. Чтобы не “ломать” структуры программы, что может не только ухудшить эффективность кода, но и изменить семантику, а также привести к висячим ссылкам, все преобразования на фазе анализа производятся над этими клонами. Описание клонов содержит изменения в сигнатуре, в теле функции, и указатель на вызывающую функцию для функций, которые должны быть встроены. По сути, на фазе WPA над графом вызовов производятся полноценные оптимизирующие преобразования, но изменения представляются графом вызовов и клонами, а сами функции остаются нетронутыми. Потом, на фазе ГМК, из клонов строятся полноценные функции, и граф вызовов перестраивается согласно аннотациям.

Также в GCC были введены так называемые *функции перехода*. Каждая функция перехода описывает значение параметра определенного вызова функции. Эта сущность была введена для таких оптимизирующих преобразований, как межпроцедурное распространение констант, встраивание и девиртуализация.

Оптимизации также производятся на всех трех этапах, но на первом и последнем – над промежуточным представлением, на втором – над аннотациями, графом вызовов, а также над клонами. На первом этапе проводятся такие оптимизации, как распространение констант, удаление мертвого кода, простые межпроцедурные оптимизации в пределах модуля, создание статического профиля, разбор дополнительных атрибутов функций (таких, как nonreturn, nonthrow и пр.), разбиение функций и некоторые другие.

5.2 Классификация функций и распространение профиля

В фазе анализа важнейшей частью является процедура распространения профиля. Функции разбиваются на 4 класса по частоте исполнения: *горячие*, *нормальные*, *исполняемые единожды* и *маловероятно исполняемые*. Эта информация нужна, чтобы определить, оптимизировать размер или быстродействие функций. Так, горячие и нормальные функции оптимизируются для быстродействия, тогда как исполняемые единожды функции оптимизируются только внутри циклов, а маловероятно исполняемые – по размеру. Если профиль исполнения недоступен, этот компиляторный проход распространяет статическую информацию следующим образом:

- Когда все вызовы функции маловероятно исполняемые – то есть вызывающая функция вероятно неисполняемая или функция холодная согласно профилю, – функция помечается как маловероятно исполняемая.
- Когда все вызывающие функции маловероятно исполняемые или исполняемые единожды, и вызывают рассматриваемую функцию по одному разу каждая, то функция помечается как исполняемая единожды.
- Когда все вызывающие функции вызываются только в момент запуска программы, то сама функция тоже помечается как вызываемая только в момент запуска.

Горячие функции помещаются в специальную подсекцию текстового сегмента файла .text.hot. Маловероятные функции помещаются в подсекцию .text.unlikely, а исполняемые при запуске – .text.startup.

5.3 Основные проблемы WHOPR-компиляции больших приложений

Несмотря на то, что в GCC применяется множество механизмов для уменьшения потребления памяти (использование только аннотаций и графа вызовов вместо промежуточного представления), потребление памяти компилятором всё еще высоко. Так, для компиляции теста cc1 требуется в 50 раз больше памяти, чем размер файла. Для компиляции же браузера Firefox нужно в 130 раз больше памяти, чем занимают все вместе взятые его исходные файлы. GCC делает отображение на память во время чтения, а также многократно открывает и закрывает файлы для считывания (в отличие от SYZYGY, где чтение и запись строго контролируются и их количество ограничено). Также представление высокоуровневых типов в памяти занимает очень много места (3.7 Гб для Firefox)[8]. Проблема необходимости перекомпиляции при изменении всего одного файла также пока открыта.

6. Google's LIPO (Lightweight Inter-Procedural Optimization)

Схема работы компиляторных инструментов от Google[10] несколько отличается от предшественников. Так как разработчики сделали упор на оптимизации с использованием профиля – а именно, встраивание функций и продвижения неявных вызовов, – они сделали процедуру сбора профиля неотъемлемой частью системы. Вместо традиционной цепочки ГПК – анализ – ГМК, в LIPO две последние фазы заменены на компоненту динамического анализа времени исполнения программы и оптимизирующий компилятор соответственно.

ГПК обрабатывает по одному файлу и генерирует инструментированный код непосредственно в исполняемом формате, без какого-либо промежуточного представления. Для того, чтобы можно было построить граф вызовов прямо во время исполнения, на вызовы функций добавляется дополнительное инструментирование.

Runtime компонента включает в себя инструмент запуска инструментированной программы, профилировщик, анализатор и генератор аннотаций. Межпроцедурный анализ начинает работу сразу после исполнения программы, происходит динамическое построение графа вызовов, причём для неявных вызовов участки графа строятся по результатам профилирования, а для явных используются результаты, собранные с помощью дополнительного инструментирования, упомянутого ранее. Анализатор изучает граф вызовов и профиль и разделяет для улучшения встраивания модули на кластеры, или супермодули, посредством жадного алгоритма. Например, если функция содержит "горячий" вызов другой функции из другого файла, то в кластер, относящийся к файлу с вызывающей функцией, будет добавлен файл с вызываемой функцией. После кластеризации дублирующие и неиспользуемые функции удаляются из кластеров, после чего на основе структуры кластера строятся аннотации. По окончании анализа суммарные данные профилирования и результаты анализа записываются в файл профиля, а аннотации – прямо в раздел данных программы для непосредственного использования в следующем этапе.

На этапе оптимизаций компилятор читает данные профиля и аннотации, а также данные о кластерах. Если в кластер входит несколько модулей, то открываются дополнительные модули и производится встраивание и продвижение неявных вызовов прямо на исполняемом коде.

6.1 Сборка

При первичной сборке проекта с использованием LIPO всё происходит по вышеописанной схеме ГПК – исполнение и анализ – оптимизация. Если же изменилось несколько файлов с исходным кодом, необходимо компилировать заново все супермодули, в которые входят измененные файлы. Поддержка

повторной компиляции реализована как отдельный инструмент к компилятору.

На тестах SPEC2006 программа показала улучшение производительности на 4.4%, при увеличении размеров файлов 25% и издержками 32% на фазе исполнения инструментированного кода и анализа. При этом большая часть издержек приходится на инструментирование, тогда как анализ по времени в среднем занимает лишь 1%, что удивительно быстро по сравнению с другими средствами межпроцедурных оптимизаций. Но, к сожалению, из-за отсутствия промежуточного представления, проводить другие оптимизации становится проблематично, и это ограничивает потенциал такого подхода к оптимизациям.

7. LLVM

Инфраструктура LLVM (Low-Level Virtual Machine – низкоуровневая виртуальная машина)[9] и все его компиляторные компоненты используют промежуточное представление – биткод – набор RISC-подобных трёхадресных инструкций. Процесс сборки программы посредством LLVM также составляют три стандартные фазы. Типичный жизненный цикл обычной сборки с межпроцедурными оптимизациями выглядит так, как показано на рисунке 5.

Цепь состоит из двух инструментов: компилятора Clang и утилиты ld с подключенным к нему плагином Gold. Здесь межпроцедурные оптимизации происходят благодаря работе Gold-плагина, а Clang производит локальные оптимизации.

Так как система предназначена для оптимизации программ среднего и малого размера, на заключительной фазе загружается вся программа помодульно, производится связывание модулей в промежуточном представлении, после чего над полученным промежуточным кодом проводятся анализ и оптимизационные преобразования по многопроходной схеме. После оптимизации производится генерация машинного кода.

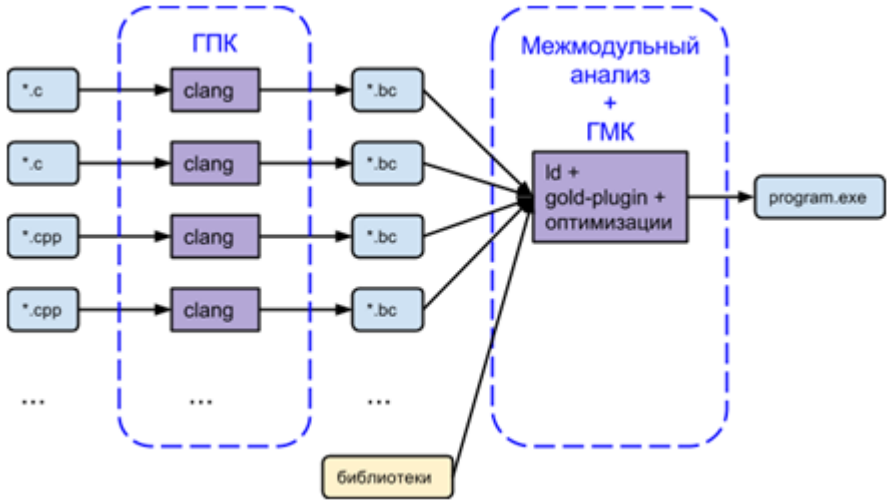

Рис. 5. Схема работы LLVM

В разрезе масштабируемости в системе LLVM присутствует ряд проблем. Промежуточное представление LLVM делится на модули, которые состоят из функций, представленных, в свою очередь, базовыми блоками, которые состоят из инструкций. Анализирующие и оптимизирующие проходы подразделяются соответственно уровню иерархии промежуточного представления на модульные, проходы функций, циклов и базовых блоков. Чаще всего используются первые два, при этом они могут запросить любые данные по иерархии ниже. Стандартная многопроходная схема последовательно работает над скомпонованной программой, при этом за модуль обработки принимается вся программа – а это значит, что проходы, работающие над модулем, могут потребовать любую часть программы во всей иерархии промежуточного представления. Это значит, что в памяти необходимо сохранять программу целиком. По этой же причине последний этап сложно проводить параллельно.

7.1 Работа над масштабируемостью LLVM

Для возможности управления потреблением ресурсов мы приняли решение сделать следующее:

- Разделить и распараллелить последнюю стадию этапа сборки программы. Вероятнее всего, разделив модули по группам с учетом зависимости по вызовам, будет возможно сделать оптимизации времени связывания более эффективными, как это происходит, например, в GCC.

- Реализовать ленивую загрузку функций на стадии оптимизации. В данный момент «лениво» функции загружаются только во время считывания файла. Оптимизации не начнутся, пока не будут загружены и скомпонованы все модули программы.
- Подстроить некоторые оптимизирующие преобразования под использование частичных данных о биткоде – аннотаций. Также необходимо собирать аннотации во время стадии ГПК и считывать для анализа.

Мы выполнили распараллеливание оптимизаций времени связывания для LLVM, разделив файлы жадным алгоритмом на количество групп, равное числу процессоров на машине. Суммарный размер в байтах каждой группы мы брали приблизительно одинаковым. Затем над группами мы провели оптимизации независимо и параллельно, только затем скомпоновали. Работы по распараллеливанию велись на основе разработанного ранее в ИСП РАН двухфазного компилятора[13].

Необходимость загружать тела функций «лениво» - то есть по мере требования оптимизациями, - родилась из того факта, что пиковая нагрузка на память приходится на момент окончания загрузки всех модулей, составляющих компилируемую программу, и далее почти не меняется. Загрузка кода во время оптимизаций также возможна за счет того, что далеко не все оптимизации времени связывания требуют код вплоть до инструкций: некоторым нужен только граф вызовов и описатели глобальных значений (например, для интернализации), некоторым – только формат данных и глобальные значения (например, для слияния констант) и т.д. Таким образом, реализовав ленивую загрузку тел функций, можно распределить нагрузку на память почти на всё время работы конвейера преобразований.

Необходимость использования аннотаций продиктована несколькими причинами: увеличением эффективности ленивой загрузки и возможностью в дальнейшем вовсе отделить фазу анализа и оптимизации от связывания и генерации машинного кода. В первом случае увеличение эффективности отложенной загрузки достигается за счет максимального оттягивания необходимости загрузки тела функций: посчитав на этапе генерации промежуточного представления такие данные, как граф вызовов, количество инструкций и константные значения, можно провести до трети от всего количества преобразований без использования тел функций.

Мы реализовали аннотации как новый тип блока в биткоде – FUNCTION_DESCRIPTOR_BLOCK_ID. Такой блок присутствует перед каждым телом функции и содержит записи, которых может быть несколько видов: запись размеров, запись с количеством значений и запись с описателями вызовов функций из текущей функции. Эти аннотации с незначительными дополнениями в коде оптимизирующих проходов позволяют продвинуться на 4 прохода в стандартном оптимизирующем наборе оптимизаций времени связывания без загрузки тел функций.

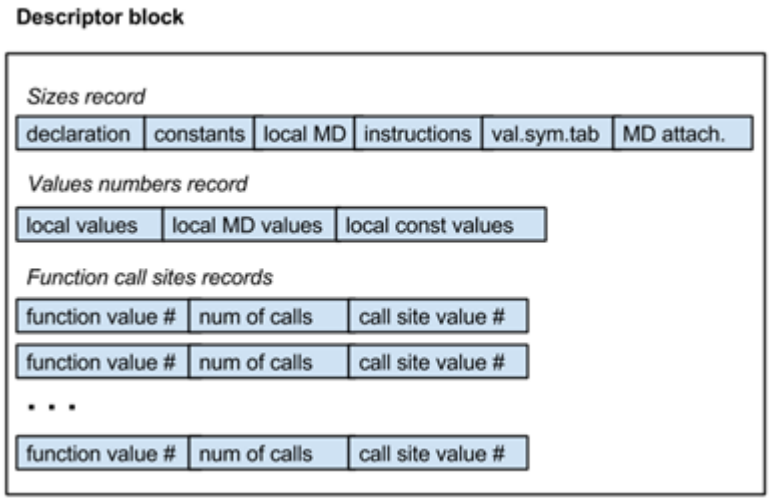

Рис. 6. Расположение данных в описателе функций

8. Заключение и результаты работы

Большинство коммерческих компиляторов не отличаются от описанной во введении модели и очень часто имеют в своей основе один из описанных выше систем. Например, межпроцедурный компилятор от Intel очень похож на WHOPR от GCC[11]. В заключение представим сводную таблицу по возможностям масштабирования всех описанных в статье систем.

Заметим, что три этапа ГПК - анализ - ГМК присутствуют во всех системах в том или ином виде. От стандартного подхода отличается только компилятор от Google, где фаза анализа заменена на анализатор времени исполнения, тем не менее, её конечным назначением тоже является межпроцедурный анализ. Системы развивались от сложных механизмов отгрузки на диск в сторону сокращения объемов необходимой межпроцедурной информации, облегчения промежуточного представления вплоть до полного отказа от него. Между тем, отказ от использования промежуточного представления сильно урезает возможности оптимизаций, усложняет анализ и саму инфраструктуру, хотя и сокращает время компиляции и объем промежуточных файлов. Если говорить о подходе LLVM – то есть об использовании легковесного промежуточного представления, – то текущая политика компилировать единый файл для оптимизаций лишает возможности проводить ГМК в параллельном режиме, хоть и вдобавок избавляет от необходимости в аннотациях. Заметим, что в LLVM на данный момент отсутствует механизм регулирования потребляемой памяти, что затрудняет компиляцию больших приложений.

	Параллелизуемые участки	**Масштабирование по памяти, способ**	**Промежуточное представление / формат файла**	**Возможность использовать повторно аннотации/ профиль**	**Увеличение размера промежуточного файла**	**Увеличение времени на оптимизацию**
HP's HLO	ГПК, ГМК	Модель «не всё в памяти»: сжатие и отгрузка на диск	IL для HP-UX / IELF	Нет / да	>x 10	x2.5-6
SYZYGY	ГПК, ГМК	Ограничение количества открытых файлов, ограничения промежуточного представления, находящегося в памяти	IL для HP-UX / IELF	? / ?	>x 5	x1.2-2.3
Open64	ГПК, ГМК	Работа только с аннотациями и графом вызовов	WHIRL / fat ELF	? / ?	x 1.7	?
GCC's WHOPR	ГПК, ГМК	Работа только с аннотациями и графом вызовов	GIMPLE / fat ELF	Нет / нет	x 1.7	x0.8
LLVM	ГПК	Облегчённое промежуточное представление, отсутствие аннотаций	LLVM / любой объектный формат	- / нет	x 1.25	-
Google's LIPO	ГПК, ГМК	Отсутствие промежуточного кода, анализ производится во время исполнения, только динамический анализ	Нет / ELF	Да / да	x 1.25	x1.32

Табл. 1. Сравнение систем межмодульных оптимизаций

Однако до сих пор узким местом остался межпроцедурный межмодульный анализ, требующий всего графа вызовов. Подход разработчиков Google строить граф динамически и пользоваться только “фактическим графом вызовов” – а не статически построенным полным графом вызовов, – облегчает анализ, но вводит в необходимость стадию промежуточного прогона программы, для которой необходимо подавать близкие к реальному использованию данные, чтобы получить реалистичные граф вызовов и

профиль. Во время промежуточного прогона нагруженная инструментированием программа будет работать в несколько раз медленнее.

8.1 Результаты тестирования масштабированной версии LLVM

На данный момент получены результаты разбиения модулей на группы и их распараллеливания. Тестирование проводилось на тестах Cray, Evas, Coremark, Lame, Clucene и Lzma с использованием архитектуры x86-64.

Имя теста	**Компоновка в 1 поток**	**Компоновка в 4 потока**	**Ускорение сборки**
Cray	0.349s	0.307s	12%
Evas	4m49.425s	3m2.489s	37%
Coremark	0.381s	0.425s	-11%
Lame	23.466s	22.537s	4%
Clucene	51.900s	51.548s	<1%
Lzma	15.3s	14.6s	4%
Smallpt	0.35s	0.32s	8%

Табл. 2. Результаты тестирования сборки с разбиением на группы для связывания

Такое заметное ускорение теста Evas связано прежде всего с его большим размером и большому количеству файлов исходного кода. Проседание производительности на smallpt связано же, наоборот, с тем, что тест очень маленький.

Имя теста	**Запуск теста, собранного в 1 поток**	**Запуск теста, собранного в 4 потока**	**Ускорение работы теста**
Cray	2m48.354s	2m48.218s	-
Evas	19.039fps	19.038fps	-
Coremark	16.300s	16.230s	<1%
Lame	23.466s	22.537s	4%
Clucene	12m30s	12m21s	1%
lzma	50.31s	50.59s	-5%
smallpt	17.11s	17.13s	-1%

Табл. 3. Результаты измерения производительности тестов, собранных с разделением на группы

Как видно из таблицы, тесты в большинстве случаев не потеряли в производительности. Лишь только lzma и smallpt немного ухудшились ввиду отсутствия учета связей в модулях во время разбиения на группы.

В настоящий момент ведутся работы по модификации проходов для использования аннотаций, а также по реализации ленивой загрузки функций.

Литература

[1]. Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable cross-module optimization. SIGPLAN Not., 33(5):301–312, 1998. ISSN 0362-1340. doi:http://doi.acm.org/10.1145/277652.277745.

[2]. Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. SIGPLAN Not., 32(5):134–145, 1997. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/258916.258928.

[3]. Sungdo Moon, Xinliang D. Li, Robert Hundt, Dhruva R. Chakrabarti, Luis A. Lozano, Uma Srinivasan, and Shin-Ming Liu. SYZYGY - a framework for scalable cross-module IPO. In CGO '04: Proceedings of the international symposium on Code generation and optimization, page 65, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

[4]. Hewlett-Packard Company, "ELF Object File Format", http://devrsrc1.external.hp.com/STKT/partner/elf-64-hp.pdf.

[5]. The Open64 Compiler Suite. www.open64.net. URL http://www.open64.net.

[6]. Gautam Chakrabarti, Fred Chow. Structure Layout Optimizations in the Open64 Compiler: Design, Implementation and Measurements.

[7]. Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, Ollie Wild. WHOPR - Fast and Scalable Whole Program Optimizations in GCC. Initial Draft, 12 Dec 2007.

[8]. Taras Glek, Jan Hubička. Optimizing real-world applications with GCC Link Time Optimization. 3 Nov 2010.

[9]. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO04), March 2004.

[10]. Xinliang David Li, Raksit Ashok, Robert Hundt. Lightweight Feedback-Directed Cross-Module Optimization. CGO'10, April 24–28, 2010, Toronto, Ontario, Canada. ACM 978-1-60558-635-9/10/04.

[11]. Ануфриенко Андрей Владимирович. Межпроцедурный анализ и оптимизации (I), 29 октября 2013, http://habrahabr.ru/company/intel/blog/199112.

[12]. Принцип Парето. Википедия, Свободная Энциклопедия. http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%9F%D0%B0%D1%80%D0%B5%D1%82%D0%BE

[13]. С.С. Гайсарян, Ш.Ф. Курмангалеев, К.Ю. Долгорукова, В.В. Савченко, С.С. Саргсян. Применение метода двухфазной компиляции на основе LLVM для распространения приложений с использованием облачного хранилища. Стр. 315-326. Труды Института системного программирования РАН, том 26, 2014 г. Выпуск 1. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), DOI: 10.15514/ISPRAS-2014-26(1)-11

Overview of Scalable Frameworks of Cross-Module Optimization

Ksenia Dolgorukova
<unerkannt@ispras.ru>
ISP RAS, 25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. Most applications consist of separated modules. Optimization of such applications during building in the traditional way "compiling-linking" is reduced to a single module optimization. However, in spite of support of cross-module optimizations by many compilers full optimizations are often unacceptable for large applications because of significant time and memory consumption. Compiler frameworks that could solve these problems with regard to hardware capabilities and user demands are called scalable cross-module optimization frameworks. In this article, we aim to consider all the different approaches to the problem of scalability of with respect to resources consumed rather than inter-procedural or cross-module optimizations or their effectivenesses because of their independence of used framework. The possibility of parallelization of various stages of whole program compilation as well as ways to save memory consumed by cross-module analysis are of interest. This overview observes several compiler frameworks for general-purpose programming languages like C, C++ and Fortran. Although, quite often these frameworks are able to optimize other languages in case of existence of corresponding front-ends.

Keywords: compilers, inter-procedural optimization, cross-module analysis and optimization.

References

[1]. Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable cross-module optimization. SIGPLAN Not., 33(5):301–312, 1998. ISSN 0362-1340. doi:http://doi.acm.org/10.1145/277652.277745.

[2]. Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. SIGPLAN Not., 32(5):134–145, 1997. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/258916.258928.

[3]. Sungdo Moon, Xinliang D. Li, Robert Hundt, Dhruva R. Chakrabarti, Luis A. Lozano, Uma Srinivasan, and Shin-Ming Liu. SYZYGY - a framework for scalable cross-module IPO. In CGO '04: Proceedings of the international symposium on Code generation and optimization, page 65, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

[4]. Hewlett-Packard Company, "ELF Object File Format", http://devrsrc1.external.hp.com/STKT/partner/elf-64-hp.pdf.

[5]. The Open64 Compiler Suite. www.open64.net. URL http://www.open64.net.

[6]. Gautam Chakrabarti, Fred Chow. Structure Layout Optimizations in the Open64 Compiler: Design, Implementation and Measurements.
[7]. Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, Ollie Wild. WHOPR - Fast and Scalable Whole Program Optimizations in GCC. Initial Draft, 12 Dec 2007.
[8]. Taras Glek, Jan Hubička. Optimizing real-world applications with GCC Link Time Optimization. 3 Nov 2010.
[9]. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO04), March 2004.
[10]. Xinliang David Li, Raksit Ashok, Robert Hundt. Lightweight Feedback-Directed Cross-Module Optimization. CGO'10, April 24–28, 2010, Toronto, Ontario, Canada. ACM 978-1-60558-635-9/10/04.
[11]. Anufrienko Andrej Vladimirovich. Mezhprotsedurnyj analiz i optimizatsii (I) [Interprocedural analysis and optimizations (I)], October 29, 2013, http://habrahabr.ru/company/intel/blog/199112.
[12]. Pareto principle. Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Pareto_principle
[13]. Sergey Gaissaryan, Shamil Kurmangaleev, Kseniya Dolgorukova, Valery Savchenko, Sevak Sargsyan. Primenenie metoda dvukhfaznoj kompilyatsii na osnove LLVM dlya rasprostraneniya prilozhenij s ispol'zovaniem oblachnogo khranilishha. [Applying two-stage LLVM-based compilation approach to application deployment via cloud storage]. Trudy Instituta sistemnogo programmirovaniya RAN [Proceedings of the Institute for System Programming of RAS], vol.26, 2014, Issue 1. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print) pp. 315-326. DOI: 10.15514/ISPRAS-2014-26(1)-11.